Next Generation Sequencing and the application of parallelized *de novo* assemblies to characterize non-model species

Authors: Miriam Barnett and Jackie Lillis

May 8, 2014

Overview

Over the past decade significant advancements have been made in the field of genetics and genomics in terms of Next Generation Sequencing (NGS) and analysis. Massively parallel sequencing platforms are able to generate millions of short (50-100bp) reads resulting in gigabytes of raw data that can expand exponentially during the analysis process. Due to these advancements, biologists have been able to conduct complex experiments ranging from the characterization of causative genetic mutations conferring diseases to the characterization of pathogen resistance genes in economically valuable crops. The generation of massive amounts of data have resulted in a demand on computer scientists to answer critical questions regarding data storage, management, and manipulation.

Computational Problem

One utilization of NGS data allows researchers to develop reference genomes and transcriptomes that serve as a roadmap to biological experiments illustrating various sequence variants, mutations, and genes. De novo assembly is very computationally intensive making it an NP-hard problem as we are searching for the shortest common sequence between a set of reads. The general tree based computational approach for the construction of a *de novo* reference assembly includes breaking reads down into short kmer fragments, developing a graph of overlapping kmers, and traversing that graph to find the optimal path based on a variety of metrics. The first assemblers introduced to reconstruct *de novo* genomes were based on a variety of approaches including, prefix tree-based (2007), overlap-extension (2008) and the de Bruijn graph representation (2001) for assembly (Simpson *et al.* 2009). However, all of these approaches suffered due to computational time and memory limitations associated with single-threaded processes being conducted with a single processor (Simpson *et al.* 2009). These approaches have been modified and parallelized in a variety of ways in hopes of finding the most accurate, time efficient, and space efficient algorithm. We have developed a massively parallelized random traversal approach that searches for the longest path of overlapping kmers along the graph, representing the most contiguous assembly.

Literature Review

ABySS: A parallel assembler for short read sequence data.
Simpson *et al.* 2009

Sequencing platforms such as Illumina and SOLID are able to generate millions to billions of raw, short (50-500bp) reads for genomes megabases in size. This volume of data can complicate and confound a variety of prefix tree and single threaded de bruijn graph approaches for *de novo* assembly. Therefore, ABySS was developed which implemented a novel distributed implementation of the de bruijn graph which enabled parallel computation.

The authors addressed a variety of computational and biological problems while developing their algorithm. These problems included the ability to handle any size genome, to allow for a distributed implementation, and to ensure an accurate biological assembly is generated in terms of N50, contig size, and %correct reconstruction. In addition, following the completion of the algorithm the authors compared its accuracy and runtime to other publicly available tools, to ensure it was a contender within the bioinformatic community. Throughout the comparisons, ABySS was considered to be a competitive assembler with other publicly available software tools.

To comprehensively evaluate the ability of the algorithm to properly assembly sequence data multiple synthetic and experimental data sets were generated for testing. The complexity in terms of sequencing errors, coverage, and varying fragment sizes were all tested to fully understand the algorithms ability to handle each. Two different data sets were created, one with fixed fragment size, consisting of 36bp paired-end sequences and perfectly tiled across the reference sequence resulting in 72 fold coverage. ABySS was able to generate 1.6M contigs greater than 100bp in length with a contig N50 of 3656bp. The final assembly was 94.4% accuracy when aligned to the reference genome. The second synthetic data set had a variable length fragment size with reduced coverage, 42-fold, which resulted in a slightly lower contig N50 at 2433bp but the accuracy observed when aligning the output back to the reference genome was still high at 94.4%.

This publication contributed a few novel concepts to the field of bioinformatics in terms of graph representation and how the computation was distributed across a cluster. The novel graph representation includes unique storage of both the location of the kmer and the adjacency list. The forward and reverse complement of each kmer is encoded with {0,1,2,3} and combined using the XOR operation on their bitset representation. Kmers are then evenly distributed across the cluster by computing a unique index for each kmer that assigns it to a specific node. Adjacency information for each kmer is stored in 8-bits, one bit for the presence or absence of an edge. This compact and memory efficient graph representation enables the distribution of the algorithm across a cluster.

De novo assembly of human genomes with massively parallel short read sequencing
Li *et al.* 2009

The authors identified a downfall of the ABySS algorithm implementation and worked to find a memory efficient solution to the problem. The main shortcoming of the ABySS assembler was its inability to generate contiguous assemblies which was evident in the low N50 values. The algorithm often resulted in fragmented contigs therefore confounding physical relationships between genetic material found on these fragments. The authors generated SOAPdenovo to overcome these pitfalls with their unique implementation of the de bruijn graph and error correcting and quality control metrics prior to generating the de novo assembly. The novel contributions include modular threaded parallelization of computationally intensive steps and a variety of graph simplification/reduction steps to reduce the amount of non-ideal paths.

The assembly algorithm consists of two phases, contig assembly of single end reads followed by scaffolding utilizing the additional information contained within paired-end reads. This could be considered an advantage and disadvantage due to the cost of paired-end sequencing. During the contig assembly phase, edges are made up of read paths, all short "tips" defined as being less than 50bp are removed from the de bruijn graph. During an experimental test set this cleaning process resulted in a significant reduction in the graph size, removing 323 million tip nodes. In addition, another 402 million low coverage nodes (appearing only once) we also removed. The algorithm also requires the contig to be greater than or equal to 100bp to be reported to the user.

During the scaffolding phase of the algorithm, paired-end information is used to merge neighboring contigs into one scaffold. This process requires at least 3 sets of paired-end reads to be in agreement for two contigs to be merged. This quality metric reduces the likelihood that contigs will be merged out of random assembly and sequencing errors. Throughout the evaluation of the algorithm, the developers saw a linear increase in contig length as depth of coverage increased from 10x-30x.

The error correction steps of the algorithm within the contig assembly phase are the most time consuming (22-24h) and computationally intensive, therefore these steps were parallelized across multiple threads. The entire algorithm has a runtime of approximately 40-48 hours on a large genome. When the authors compared SOAPdenovo to ABySS they were able to observe a larger N50 value with their algorithm compared to ABySS with N50 values equalling 4611 and 1499, respectively. In addition SOAPdenovo's runtime was roughly half of ABySS but SOAPdenovo's memory consumption was greater than ABySS due to ABySS's utilization of a memory efficient graph representation.

Parallelized short read assembly of large genomes using de Bruijn graphs
Liu, Schmidt, Maskell, 2011

As previously mentioned, next-generation sequencing technologies create a dire need for constructing large genomes efficiently. The major problem, however, is the fact that the nature of this problem requires a large amount of computational resources. To date, most of the de novo assembly algorithms do not fully recognize the significant power of various parallel computing techniques. Liu, Schmidt and Maskell present a novel approach by utilizing not only shared-memory multi-core CPUs, but also distributed-memory compute clusters. The result, PASHA (parallelized short read assembler) gains efficiency and scalability.

Similar to Abyss, PASHA makes use of de Bruijn graphs. After k-mers are generated from the inputted reads, they must be compared to determine multiplicity and overlap. The authors note that this is difficult to parallelize because each k-mer must be aware of every other k-mer. Generating the graph is executed using a multi-threaded design; however, the authors fail to elaborate on the implementation. Once the graph is generated, however, steps can be performed to reduce the memory overhead by simplifying the graph. The graph is pruned by removing paths that lead quickly to dead ends; thus, the overall size is reduced and no valuable information is lost. Traversing the graph is also done in parallel. Again, however, the authors are vague about the implementation. It is noted, though, that communication between the various structures was an issue. The time required for memory allocations and de-allocations was observed to significantly decrease performance.

PASHA was evaluated against three accredited programs: Velvet, Abyss and SOAPdenovo. Three small genomes were assembled. Overall, PASHA was able to assemble larger contigs; however, of the assembled contigs, a higher percent were incorrect when compared to other programs. Moreover, PASHA was only able to cover on average 92.27% of the genomes, whereas the other three programs were able to cover on average over 97% of the genomes. Despite the less than ideal result, PASHA did have a noticeably lower run time. For example, when constructing one genome, PASHA took 325 seconds. The other three programs, however, each took over 500 seconds. It is not shown whether or not this difference in run time is observed over larger datasets.

Lastly, the authors demonstrate the strong scaling performance. As the number of cores increase, the run time for a given data set does in fact decrease. The scaling, however, does not appear to be ideal. For example, for one dataset, the program ran for roughly 150 seconds on two CPU cores and 125 seconds on 64 CPU cores.

While this publication was vague on the algorithm specifics, however, it still proved to be useful. First, this paper address some of the issues we faced, such as large memory overhead. Secondly, while the authors were able to show some scalability, results were less than ideal. This article helped us troubleshoot our scalability problem.

Program Design

Both the sequential and parallel programs can be broken down into four main sections: (1) reading in the sequencing reads from a file, (2) generating k-mers of a designated length, (3) constructing the de Bruijn graph from overlapping k-mers, and (4) traversing the graph the specified number of times in order to find the longest path representing the optimal reconstructed genome.

First, sequencing reads are taken as input from the file specified on the command line. The file must be formatted such that the first line contains the number of reads and the total length separated by a space. Following the first line, each read is introduced with a header line designated by a right angle bracket. The read follows on the line below with no other text or white-space. The reads are stored in an array as strings.

With any NGS project, there is always the possibility that an area of the genome will be missed due to random error of the sequencing technology. In order to ensure total coverage, reads are further broken down into small fragments of length 'k,' known as k-mers. The theory is that by reducing the size of the fragments, sequencing gaps can be accounted for because by chance a k-mer will cover the gap. For example, a 100-nucleotide read generates 46 overlapping 55-mers. Thus, even "if some 100-mers occurring in the genome are not generated as reads, this 'read breaking' procedure ensures that nearly all 55-mers appearing in the genome are detected" (Compeau, Pevzner, Tesler, 2011). The ideal 'k' value is dependent on the length of the total read; generally speaking it is one-third the read length.

After 'k' is determined, the k-mers are generated using a 1 base-pair sliding window. Each k-mer creates a new instance of a Nodes object having four attributes: the string representation of the sequence, the Edgelist initialized as null, hashcode for the k-mer-1, and a hashcode for the (k-1)-mer. The k-mers are again stored in an array.

In order to generate the de Bruijn graph, the k-mer nodes must be connected by edges. An edge between two nodes is created if the k-mer overlap by k-1. In order to to efficiently determine the edges, each k-mer's (k-1)-mer hashcode was compared against each other k-mer's (k-1)-mer hashcode. If the hashcodes were equal, the two k-mer strings were then compared one character at a time to ensure the match was not the result of a hash collision. The overlapping k-mers were then added to the EdgeList of the original k-mer. The EdgeList is an ArrayList containing the indexes of the k-1 overlapping k-mers in the original k-mer array.

Once the de Bruijn graph is generated, the sequential and parallel programs both traverse it in order to find the longest path representing the optimal genome. Starting at a randomly generated index, the traversal continues by randomly selecting an index from the node's EdgeList. This continues until we reach a k-mer with no children. The indexes visited as stored in an ArrayList. The traversal which visits the most nodes, signified by the largest

ArrayList, is used to rebuild the genome by adding the corresponding nucleotide substring from each visited k-mer.

Sequential Algorithm Description (DeNovoSeq)

The sequential algorithm performs the aforementioned process on one thread. The read input is contained in a while loop, such that the process continues until the file does not contain another line. Two different functions are used to generate the k-mers and build the graph. An array of Node objects with the edge adjacency lists set to null is constructed within the makeKmers method. The k-mers are then compared for overlap using the hashCodes within the compareKmers function. This function has no return value, rather, it alters the Node object's adjacency list by calling the setEdge modify function contained in the Node class. After the completion of these two steps, the graph represented as an array containing adjacency lists, is complete.

The graph is traversed the specified number of times by iterating through a for loop. traverseGraph is called, supplying the graph as an input parameter. This function utilizes a while loop to visit the adjacency list of a given node until a node is reached with no adjacency list (in other words, until a node with no children). An ArrayList holds the integer values of the indices of the nodes visited and is returned upon completion of the while loop. The length of this ArrayList is compared against the length of the prior traversals. Only the largest ArrayList of indices is kept. Finally, the buildString function builds the string representation of the genome by utilizing a for loop. The portion of the k-mer is concatenated to the string by using the getKmer accessor in the Node class.

Parallel Algorithm Description (DeNovoClu)

A large portion of the cluster program runs sequentially. The Job main method, which extends the parallel java2 Task, first performs the sequential portion. The file is read in, k-mers are generated and the graph is constructed using the same aforementioned functions. Once the graph is constructed, a graphTuple object is generated. This object has one attribute: kmers, an array of Node objects. Since this object is streamable, a writeOut and readIn method are present. One graphTuple is put into tuple space for each core specified.

The Job main method then sets up the parallel portion by calling the masterFor for the specified number of traversal. masterFor calls the workerTask, a nested class which again extends Task. Within the workerTask, the workerFor loop executes the traversal. In the parallel version, however, the traverseGraph does not return an ArrayList. Rather, it returns a ReductionVbl. This ReductionVbl extends Tuple and implements Vbl. The attribute, traversal, is an ArrayList of integers, which holds the indices visited during the traversal. Again, since this is a tuple, the writeOut and readIn methods are necessary. Since this class implements Vbl, the reduce and set method were required to be implemented. The reduce method compares to ArrayLists and keeps the larger one. The set method replaces the ArrayList with a deep copy of

another ArrayList. It is important to note that a local reduction occurs within a core, only the largest ArrayList from all threads is put back into tuple space.

Once a core finishes its specified number of traversals, the Job main method calls the ReduceTask. This reduces one by one by repetedly calling the reduce method within the ReductionVbl. It is important to note that only the longest traversal is kept, which is again based on the length of the ArrayList. Once the longest traversal is determined, the buildString function is called. This function is identical to the sequential one and builds the string representation of the genome.

Manual Overview

The parallel and sequential programs included within the provided jar file can be run on RIT's cluster, Tardis (tardis.cs.rit.edu). The following section will go over proper file extraction, compilation, and command line execution. In addition to all java code, all synthetic test cases are also included within the java file for necessary experimental replication. A complete list of test cases is included in the appendix, including brief instruction for the generation of new test cases.

Developer's Manual

(1) Since these programs need to be able to access pj2 the proper classpaths must initially be set. For more detailed instructions, please reference course website:
(http://www.cs.rit.edu/~ark/runningpj2.shtml):

```
export CLASSPATH=.:/var/tmp/parajava/pj2/pj2.jar
export LD_LIBRARY_PATH=/var/tmp/parajava/pj2:$LD_LIBRARY_PATH
```

(2) Extract and compile all necessary files contained within the provided jar file, DeNovo.jar

```
/usr/local/dcs/versions/jdk1.7.0_11_x64/bin/jar xvf DeNovo.jar
/usr/local/dcs/versions/jdk1.7.0_11_x64/bin/javac *.java
```

User's Manual

(1) Execution of Sequential program:

```
/usr/local/dcs/versions/jdk1.7.0_11_x64/bin/java pj2 jar=DeNovo.jar DeNovoSeq
<File_of_Reads> <Number_of_Traversals>
```

- <File_of_Reads> is a *.fasta or *.txt of reads with the total number of reads and the length of total reference for assembly listed on the first line. For example:
  450 500
  >Read_1
  ATGCG…
- <Number_of_Traversals> is a user defined number that controls how many traversals are performed on the graph of overlapping kmers.

Expected Sequential Output:

```
The best traversal result is:
AACTGCTCAATCCCTCCATATTCACAACCAATGTACCAAAACAATTTTGGAGAGATGCTATCTTAGCAGCTACTTATT
TAATTAGTCGTCTGCCAAGCCAAGTCTTAAATTATCAAACACGACTTGACCATTTGCTCTATGTTTTCCCTCACATCC
GAACACTCACTTCCATACCGAAAACAGTCTTTGGTTGCACTGTTTTTGTCCATAATTAGTGTTAATAAAAGTAAACTC
GATCCTAGGGTAATTAAGTGTATGTTTCTTGGATACTCTCCCACTCAAAAAGGTTATTGTTGTTAGATCACCAAGAAA
TTCTACACTTCTTTAGATGTTACTTTCTTTGAATCCCAACCTTATTACTAAAAATTCCTTCAGGAAGAGACATCAAGT
GAAGCTAATTTTTAGGAAACACTTGTTCTCCATCAGTCCAAAGTCTCACAATCTGTTCCTCGGTCTGTTGTTCCTCAG
TCCGATGTCAATAAGAGTCACAATTCTCATGTCCTTCCTGATTCGGGTTCCCTATTGTGTTCTCCCACACCAAAGTCC
TCACAACA

The length is:
554

Took 0.467865501 s
```

The program writes the assembled reference sequence (string ATCG), total length, and runtime (seconds) to standard output.

(2) Execution of Cluster Parallel program:

```
/usr/local/dcs/versions/jdk1.7.0_11_x64/bin/java pj2 jar=DeNovo.jar DeNovoClu
<File_of_Reads> <Number_of_Traversals> <cores>
```

- <File_of_Reads> is a *.fasta or *.txt of reads with the total number of reads and the length of total reference for assembly listed on the first line. For example:
  450 500
  >Read_1
  ATGCG…
- <Number_of_Traversals> is a user defined number that controls how many traversals are performed on the graph of overlapping kmers.
- <cores> user defined amount of cores for this program to distribute traversals across

Expected Parallel Output:

```
Sequential RunTime 318
No Tracker at localhost:20618; job will run in this process
Job 1 launched Tue May 06 22:58:07 EDT 2014
Job 1 started Tue May 06 22:58:07 EDT 2014
The length is: 570
TCAAAAAGAAAAAATCCGTCATTTACTTGAAACTGCTCAATCCCTCATATTCACAACCAATGTACCAAAACAATTTTG
GAGAGATGCTATCTTAGCAGCTACTTATTTAATTAGTCGTCTGCCAAGCCAAGTCTTAAATTATCAAACACGACTTGA
CCATTTGCTCTATGTTTTCCCTCACATCCGAACACTCACTTCCATACCGAAAACAGTCTTTGGTTGCACTGTTTTTGT
CCATAATTAGTGTTAATAAAAGTAAACTCGATCCTAGGGTAATTAAGTGTATGTTTCTTGGATACTCTCCCACTCAAA
AAGGTTATTGTTGTTAGATCACCAAGAAATTCTACACTTCTTTAGATGTTACTTTCTTTGAATCCCAACCTTATTACT
AAAAATTCCTTCAGGAAGAGACATCAAGTGAAGCTAATTTTTAGGAAACACTTGTTCTCCATCAGTCCAAAGTCTCAC
AATCTGTTCCTCGGTCTGTTGTTCCTCAGTCCGATGTCAATAAGAGTCACAATTCTCATGTCCTTCCTGATTCGGGTT
CCCTATTGTGTTCTCCCACACCAAAGTCCTCACAACAATA
Job 1 finished Tue May 06 22:58:07 EDT 2014 time 61 msec
```

Algorithm Evaluation

The algorithm was rigorously tested with a variety of data sets ranging in size and complexity across multiple cores with a varying number of traversals. Test cases were generated to properly characterize the strong and weak scaling demonstrated by the algorithm. In addition, multiple rounds of testing were executed to gain insight into the impact that increased traversals can have on the length of assembly, the impact that larger problems can have on sequential runtime, and the proportion of time spent within the sequential and parallel processes of the algorithm. This level of evaluation enabled us to draw multiple conclusions regarding performance.

The proportion of computational time spent within the sequential and parallel sections of the algorithm was evaluated using a 1kb reference data set consisting of 950 reads generated using a 1bp sliding window. The sequential and parallel runtimes, in msec, were recorded following program execution on 1 to 4 cores. Overall, data suggests that the majority of runtime is spent within the sequential portion of the code, reading in the file of reads, generating the kmers, and building the overlapping kmer graph data structure (Figure 1). In addition, these data suggest that as we increase the number of cores our time spent within the parallel portion of the code slightly increases (Figure 1).

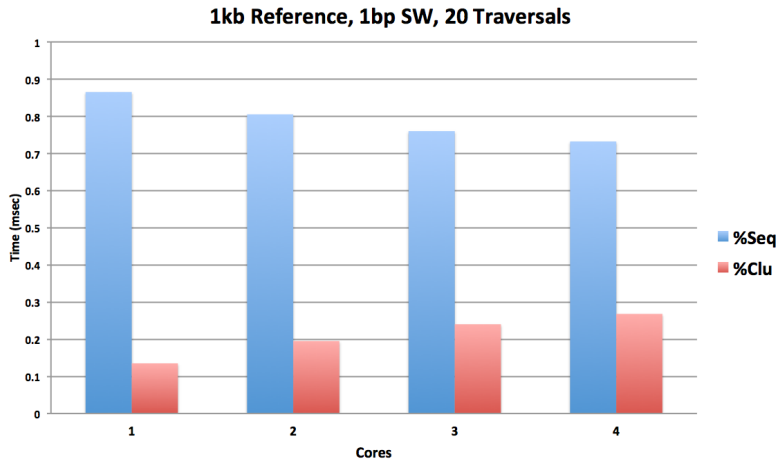

Figure1. Illustrates the proportion of computational time spent (msec) within both the sequential and parallel regions of the program across 1-4 cores on Tardis. The majority of runtime is spent sequentially, ranging from ~74-85% of the programs total runtime. As the amount of cores increase, we observe a slight increase of time spent in parallel.

One main goal of any de novo assembler is to *fully* and *accurately* assemble a given set of reads to include all genes, regulatory elements, and intronic regions for usage within genetic and genomic experiments. Therefore, it is ideal to see a high percentage of the reference material assembled with a low frequency of sequencing and assembler errors such as insertions and deletions. To characterize our assembler's ability to accurately and completely assemble a reference sequence given a set of reads, a variety of user defined parameters were evaluated. Synthetic data sets were generated using a variable length sliding window (1,2,5) to illustrate the performance of the algorithm of ideal and less than ideal depth of coverage. Results indicate that there is not a significant difference in the ability of the algorithm to assemble the reference sequence when a higher order sliding window is applied during the read generation process (Figure 2). The overall accuracy of the assembler is not decreased as the sliding window is increased (Figure 2). In addition, the size and complexity of the synthetic data set were held constant and and the length of the final assembly was recorded while increasing the user defined number of traversals (20,100,1000). It was initially hypothesized that as the number of traversals increase, the more likely we are at finding the optimal assembly, defined as the longest genome. The data supports this hypothesis and shows a significant improvement in the length of the assembly as you increase the number of cores from 20 to 1000 (Figure 3).

```
CLUSTAL 2.1 multiple sequence alignment


500_sw2         --------------------------------TTTGGAGA-GATGCTATTCTTAGCAGCT
500_sw1         ------------------------AAAACAATTTTGGAGAAGATGCTAT-CTTAGCAGCT
500_sw5         CTCATATTCACAACCAAATGTACCAAAACAATTTTGGAGA-GATGCTAT-CTTAGCAGCT
                                        ******** ******** **********

500_sw2         ACTTATTTAATTAGTCGTCTGCCAAGCCAAGTCTTAAATTATCAAACACGACTTGACCAT
500_sw1         ACTTATTTAATTAGTCGTCTGCCAAGCCAAGTCTTAAATTATCAAACACGACTTGACCAT
500_sw5         ACTTATTTAATTAGTCGTCTGCCAAGCCAAGTCTTAAATTATCAAACACGACTTGACCAT
                ************************************************************

500_sw2         TTGCTCTATGTTTTCCCTCACATCCGAACACTCACTTCCATACCGAAAACAGTCTTTGGT
500_sw1         TTGCTCTATGTTTTCCCTCACATCCGAACACTCACTTCCATACCGAAAACAGTCTTTGGT
500_sw5         TTGCTCTATGTTTTCCCTCACATCCGAACACTCACTTCCATACCGAAAACAGTCTTTGGT
                ************************************************************

500_sw2         TGCACTGTTTTTGTCCATAATTAGTGTTAATAAAAGTAAACTCGATCCTAGGGTAATTAA
500_sw1         TGCACTGTTTTTGTCCATAATTAGTGTTAATAAAAGTAAACTCGATCCTAGGGTAATTAA
500_sw5         TGCACTGTTTTTGTCCATAATTAGTGTTAATAAAAGTAAACTCGATCCTAGGGTAATTAA
                ************************************************************

500_sw2         GTGTATGTTTCTTGGATACTCTCCCACTCAAAAAGGTTATTGTTGTTAGATCACCAAGAA
500_sw1         GTGTATGTTTCTTGGATACTCTCCCACTCAAAAAGGTTATTGTTGTTAGATCACCAAGAA
500_sw5         GTGTATGTTTCTTGGATACTCTCCCACTCAAAAAGGTTATTGTTGTTAGATCACCAAGAA
                ************************************************************

500_sw2         ATTCTACACTTCTTTAGATGTTACTTTCTTTGAATCCCAACCTTATTACTAAAAATTCCT
500_sw1         ATTCTACACTTCTTTAGATGTTACTTTCTTTGAATCCCAACCTTATTACTAAAAATTCCT
500_sw5         ATTCTACACTTCTTTAGATGTTACTTTCTTTGAATCCCAACCTTATTACTAAAAATTCCT
                ************************************************************

500_sw2         TCAGGAAGAGACATCAAGTGAAGCTAATTTTTAGGAAACACTTGTTCTCCATCAGTCCAA
500_sw1         TCAGGAAGAGACATCAAGTGAAGCTAATTTTTAGGAAACACTTGTTCTCCATCAGTCCAA
500_sw5         TCAGGAAGAGACATCAAGTGAAGCTAATTTTTAGGAAACACTTGTTCTCCATCAGTCCAA
                ************************************************************

500_sw2         AGTCTCACAATCTGTTCCTCGGTCTGTTGTTCCTCAGTCCGATGTCAATAAGAGTCACAA
500_sw1         AGTCTCACAATCTGTTCCTCGGTCTGTTGTTCCTCAGTCCGATGTCAATAAGAGTCACAA
500_sw5         AGTCTCACAATCTGTTCCTCGGTCTGTTGTTCCTCAGTCCGATGTCAATAAGAGTCACAA
                ************************************************************

500_sw2         TTCTCATGTCCTTCCTGATTCGGGTTCCCTATTGTGTTCTCCCACACCAAAGTCCTCACA
500_sw1         TTCTCATGTCCTTCCTGATTCGGGTTCCCTATTGTGTTCTCCCACACCAAAGTCCTCACA
500_sw5         TTCTCATGTCCTTCCTGATTCGGGTTCCCTATTGTGTTCTCCCACACCAAAGTCCT----
                ************************************************************

500_sw2         ACAATATGT--
500_sw1         ACAATATGTAA
500_sw5         -----------
```

Figure 2. Compares the optimal assemblies generated by DeNovoClu for three different test cases all based on a 500bp reference sequence, differing in the sliding window used during read generation*_sw1|2|5.

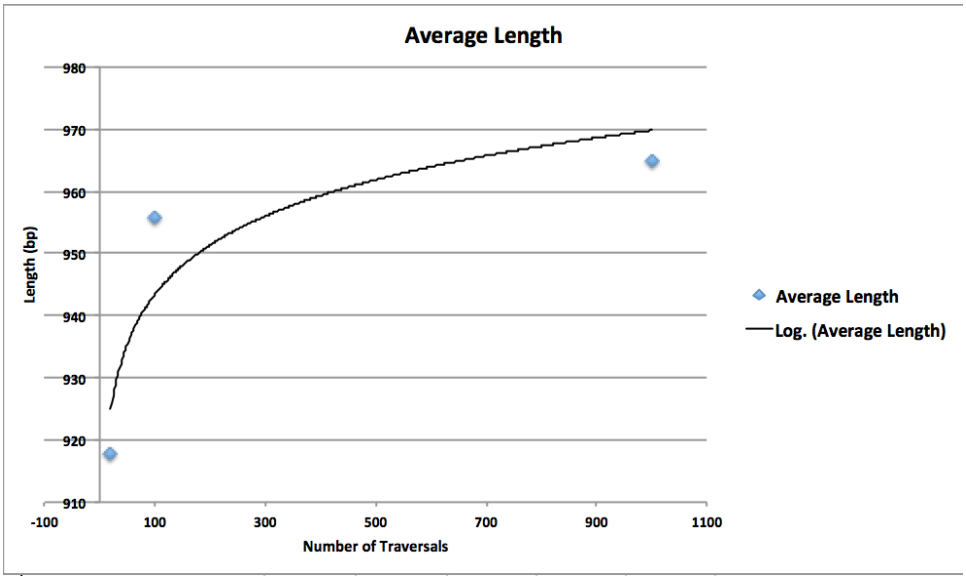

Figure 3. Illustrates the impact that an increased number of traversals can have on the length of the final assembly. The scatter plot displays the average length in base pairs (bp) at 20, 100 and 1000 traversals with the logarithmic trend line further illustrating the positive relationship between traversals and length.

Strong Scaling

Strong scaling can be defined as measuring a parallel program's performance across an increasing number of cores while holding the problem size constant. In an ideal strong scaling situation, a 1/K speedup is observed where K is the number of cores being evaluated (Kaminsky's Book Ref). Multiple data sets, ranging in size and complexity, were generated to evaluate the strong scaling demonstrated by our parallel program (Table 1). The data sets size remained the same while the number of cores utilized were increased from 1 to 4. A speedup of 1/K was never observed for any of our six test cases, instead a slight increase in runtime was observed while increasing the number of cores for five of the six test cases(Figure 4). Increasing the number of cores did slightly decrease the overall runtime for our test data set consisting of 90 reads for the assembly of a 500bp reference sequence with a 5bp sliding window (Figure 4). For this example, the average length of the de novo assembly was 435bp, 87% of the total expected length of assembly. This average length was the smallest assembly, compared to the results of the 500bp test sets with 1 and 2bp sliding windows with average lengths of 458bp and 429, respectively. Therefore, although we had achieved slight scaling with the data set, there was a trade-off in terms of the percent of reference assembled.

Table 1. Displays the information regarding the size, complexity, and number of traversals used to evaluate the strong scaling of our parallel program across 1-4 cores on Tardis

Reference Size (bp)	Sliding Window	Number of Traversals
500	1	20
500	2	20
500	5	20
1000	1	20
1000	2	20
1000	5	20

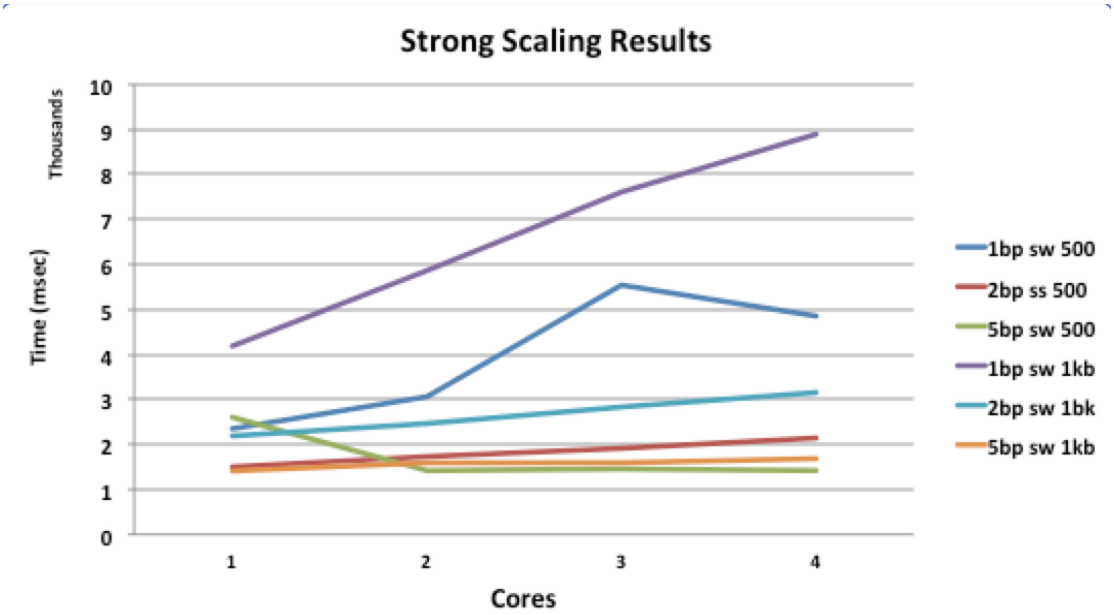

Figure 4. Displays the impact of increasing the number of cores on parallel runtime while keeping problem size constant (msec). Six different test cases were evaluated as detailed in table 1. Slight strong scaling was observed for the smallest test case, 500bp with a 5bp SW.

Non-ideal Scaling

The performance of our algorithm displayed non-ideal strong scaling. Only one of our test cases demonstrated slight scaling while the number of cores increased, but we cannot conclude with any level of confidence that this speedup is significant. The results of our testing suggest that there could be multiple reasons for the lack of scalability. For instance, a great deal of computational time is spent within the sequential portion of our algorithm by reading in the

reads, creating kmers, and building the graph structure (Fig.1). The time spent executing the traversals is so minimal, that distributing them across a cluster instead increases the amount of computational time due to the time spent transferring the graph tuple in/out of tuple space. In the future, it might be possible to parallelize the graph generation; however, a shared-memory system might be necessary.

Moreover, the amount of information put into tuple space was not efficient. The graph tuple contains a lot of unnecessary information including, the k-mer hashcodes, k-mer string sequences, and adjacency lists. The adjacency list information is the only required data needed during the parallel traversals. This is drastically slowing down the parallel computations. In the future, the tuple should be revised to only include the adjacency list.

Lastly, it is probable that the fixed schedule created an unbalanced load. If one thread was traversing through a long path and another thread was traversing through a very short path, the threads would spend an unequal amount of time executing the parallel loop. To remedy this situation, a symmetric multiprocessor (SMP) parallel de novo assembly was coded. The run time for fixed, dynamic and guided schedules were compared, as seen in table two. While the dynamic schedule slightly reduced the run time, the program still failed to ideally scale, as seen by the insignificant reduction as the number of threads increased.

Table 2. Observed effect of scheduling on DeNovoSmp run time. For the dynamic schedule with 100 traversals, a chunk size of 15 was determined to be ideal.

schedule	threads	time
fixed	5	156.86
	10	155.67
guided	5	155.37
	10	151.98
dynamic	5	146.36
	10	145.98

Weak Scaling

In contrast to strong scaling, weak scaling can be evaluated by increasing both the number of cores for parallelization and the problem size. Five different test cases were create for the evaluation of weak scaling for our parallel program (table 3). Increasing the test case information as we increase our cores does increase our time spent in the sequential portion of the algorithm (Figure 5). As defined in the text, sizeup is a metric that can be used to evaluate a

program's weak scaling performance. This metric takes into account the ratio of time spent within the sequential and parallel computation and is defined below in Equation 1 (Kaminsky, 2014). We did not observe any ideal weak scaling with the implementation of our algorithm (Figure 5).

$$Sizeup(N,K) = \frac{N(K)}{N(1)} \times \frac{T_{seq}(N(1),1)}{T_{par}(N(K),K)}$$

Equation 1. Sizeup can be used to evaluate the weak scaling performance of an algorithm taking both sequential and parallel performance into account where N is the size of the problem set and K is the number of cores. Notice the numerator contains information regarding the Time (T) spent in sequential whereas the denominator is T spent in parallel (Kaminsky, 2014: Chapter 9 Equation 9.3).

Table 3. Displays the information regarding the size, complexity, and number of traversals used to evaluate the weak scaling of our parallel program across 1-4 cores on Tardis

Reference Size (bp)	Sliding Window	Number of Traversals
600-900	1	20
600-900	2	20
600-900	5	20
1000-4000	2	20
1000-4000	5	20

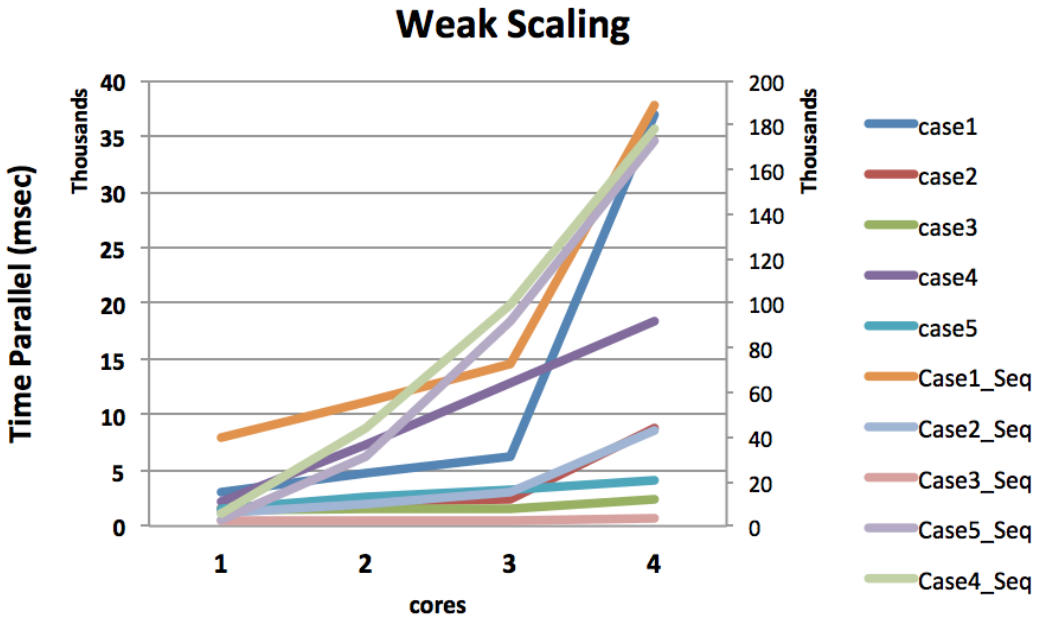

Figure 5. Displays the impact of increasing the number of cores on sequential and parallel runtime while also increasing the problem size (msec). The primary axis shows time spent in the parallel portion, whereas the secondary axis illustrates time spent in the sequential portion. Five different test cases were evaluated as detailed in table 2. Non-ideal weak scaling was observed in all test cases.

Non-ideal Scaling

Our algorithm does not demonstrate strong or weak scaling. The reason for this has been previously discussed but resides in the fact that the sequential portion of our algorithm is the most computationally intensive section of the program and therefore cluster parallelization does not result in any speedup due to the trafficking of graph tuples in and out of tuple space. Gustafson stated that as a problem size increases it is the parallel portion of the algorithm that increases, not the sequential portion (Kaminsky, 2014). Although we know that often the sequential *does* increase and we can account for this in the sizeup metric, our sequential computation is memory intensive and accounts for ~85% of our runtime. Because of this additional parallelization does not help our ability to solve the problem.

Future Direction

Currently our assembler can only reconstruct one reference sequence, representing one contig or chromosome of a genome. This is applicable for some small bacterial genomes, yet more complex genomes such as plant and animal can contain multiple chromosomes that are diploid, having two copies of every gene or even as complex as a hexaploid. Therefore, we would need to implement functionality in order to construct multiple output fragments to represent multiple chromosomes. In addition, the authors of SOAPdenovo discussed the utilization of paired-end sequencing in order to scaffold together contigs that were part of the same putative chromosome. Currently, our assembler can only utilize single end reads to assemble a final genome. Since we are not currently generating multiple contigs, our approach is valid. As we tackle larger, more complex genomes the utilization of paired-end reads can become essential for the generation of accurate and contiguous assemblies.

In addition, we discussed the causation for our non-ideal scaling to reside within the sequential portion of the algorithm involving a memory intensive graph. Before we can scale our algorithm up to more complex genomes and utilize paired-end reads we would need to look back at the ABySS and SOAPdenovo publications, since both implement measures to efficiently store the graph information and reduce graph complexity.

Knowledge Gained:

It is a difficult task to try to convey all that we learned from this project. When we first implemented our sequential algorithm, we were not concerned with efficiency. Once the algorithm was functioning correctly, we revisited it and were able to drastically decrease the run time by implementing two major changes. First, instead of comparing two k-mers one character at a time, we made use of the hashcode. If two hashcodes were the same, we then compared the k-mers one character at a time to ensure a hash collision had not occurred. Secondly, we realized that the genome string only needed to be constructed for the longest traversed path. Prior to this change, we were constructing the string for each and every traversal. Run times for the two programs can be seen below in table 4. This was the first time that we revisited code in order to improve the efficiency.

Table 4. Effect of optimizations on run time of sequential algorithm. It is evident that the run time was significantly reduced

Size of Data Set (kb)	time (s)- no improvements	time (s) - with improvements
1 kb	46.2	4.26

2 kb	136.80	12.88
5 kb	1224.56	141.88

Furthermore, we were able to expand our knowledge of lecture material. For example, we observed the effect of load balancing by implementing different schedules. We also were able to see the effect of chunk size on the dynamic schedule. When the chunk was too small or too large, the load again became unbalanced. Moreover, we were able to successfully code a reduction variable. In previous homework assignments, we just used the LongVbl. When we implemented our own Vbl, we observed the inner workings of the interface, specifically the reduce and set methods. Lastly, we gained more experience with tuple space. We learned that it is advantageous to only place necessary attributes into tuple space.

Teammate Contributions

Topic Selection: Miriam Barnett and Jackie Lillis
Topic Brainstorming: Miriam Barnett and Jackie Lillis
Sequential, SMP, Clu Coding: Miriam Barnett (Lead) Jackie Lillis (support)
Algorithm Testing: Miriam Barnett (support) and Jackie Lillis (lead)
Presentation 1: Miriam Barnett and Jackie Lillis
Presentation 2: Miriam Barnett and Jackie Lillis
Presentation 3: Miriam Barnett and Jackie Lillis
Presentation 4: Miriam Barnett and Jackie Lillis
Deliverables: Miriam Barnett and Jackie Lillis

References

Kaminsky, A. (2014). *BIG CPU, BIG DATA: Solving the World's Toughest Computational Problems with Parallel Computing.*

Liu, Y., Schmidt, B., & Maskell, D. L. (2011). Parallelized short read assembly of large genomes using de Bruijn graphs. *BMC Bioinformatics`*, 12:354

Li, R., Wang, J., Qian, W., Ruan, J., Zhu, H., Wang, J., et al. (2010). De Novo Assembly Of Human Genomes With Massively Parallel Short Read Sequencing. Genome Research, 20(2), 265-272.

Simpson, J.; Wong, K.; Jackman, S.; Schein, J.; Jones, S.; Birol, I.(2009). ABySS: A parallel assembler for short read sequence data . Genome Research, 19, 1117-1123.

Appendix

Generate Synthetic Reads:

```
* Ensure perl is functioning on your computer
** slidingWindow.pl should be contained within DeNovo.jar

perl slidingWindow.pl <Fragment_of_DNA> <Read_Size> <Length_of_DNAFragement>
<SW> <output_Reads>
```

- <Fragment_of_DNA> fasta file of 1 segment of DNA
- <Read_Size> user define size of read, e.g. 50
- <Length_of_DNAFragment> length of DNA segment given in <Fragment_of_DNA>
- <SW> user defined length of sliding window, e.g. 1,2,5
- <output_Reads> output file containing reads lay out where 450 = number of reads and 500 = <Length_of_DNAFragment>:
  450 500
  >Read_1
  ATCG...

Test Cases contained within DeNovo.jar

```
Reads_1k_2sw.txt
Reads_1k_5sw.txt
Reads_2k_1sw.txt
Reads_2k_2sw.txt
Reads_2k_5sw.txt
Reads_2kb_1sw.txt
Reads_2kb_2sw.txt
Reads_2kb_5sw.txt
Reads_3k_1sw.txt
Reads_3k_2sw.txt
Reads_3k_5sw.txt
Reads_3kb_1sw.txt
Reads_3kb_2sw.txt
Reads_3kb_5sw.txt
Reads_4k_1sw.txt
Reads_4k_2sw.txt
Reads_4k_5sw.txt
Reads_4kb_1sw.txt
Reads_4kb_2sw.txt
Reads_4kb_3sw.txt
Reads_4kb_5sw.txt
Reads_500bp_1sw.txt
Reads_500bp_2sw.txt
Reads_500bp_5sw.txt
Reads_600bp2_1sw.txt
```

```
Reads_600bp_1sw.txt
Reads_600bp_2sw.txt
Reads_600bp_5sw.txt
Reads_700bp_1sw.txt
Reads_700bp_2sw.txt
Reads_700bp_5sw.txt
Reads_800bp_1sw.txt
Reads_800bp_2sw.txt
Reads_800bp_5sw.txt
Reads_900bp_1sw.txt
Reads_900bp_2sw.txt
Reads_900bp_5sw.txt
```